Natural Language Processing

Lab 10: RNNs and LSTMs

Objective

To understand the foundational steps for implementing Recurrent Neural Networks (RNNs) and Long Short-Term Memory (LSTMs) models in Natural Language Processing (NLP) tasks, such as language modeling, sequence labeling, and text classification, enabling handling of sequential data with temporal dependencies.

Neural Network

A modern neural network is a network of small computing units, each of which takes a vector of input values and produces a single output value. Neural networks share much of the same mathematics as logistic regression. But neural networks are a more powerful classifier than logistic regression, and indeed a minimal neural network (technically one with a single 'hidden layer') can be shown to learn any function.

The building block of a neural network is a single computational unit. A unit takes a set of real valued numbers as input, performs some computation on them, and produces an output.

At its heart, a neural unit is taking a weighted sum of its inputs, with one additional term in the sum called a bias term. Given a set of inputs $\mathbf{x_1...x_n}$, a unit has bias term a set of corresponding weights $\mathbf{w_1...w_n}$ and a bias **b**, so the weighted sum **z** can be represented as:

$$z = b + \sum_i w_i x_i$$

Often it's more convenient to express this weighted sum using vector notation; recall from linear algebra that a vector is, at heart, just a list or array of numbers. This vector we'll talk about **z** in terms of a weight vector **w**, a scalar bias **b**, and an input vector **x**, and we'll replace the sum with the convenient dot product:

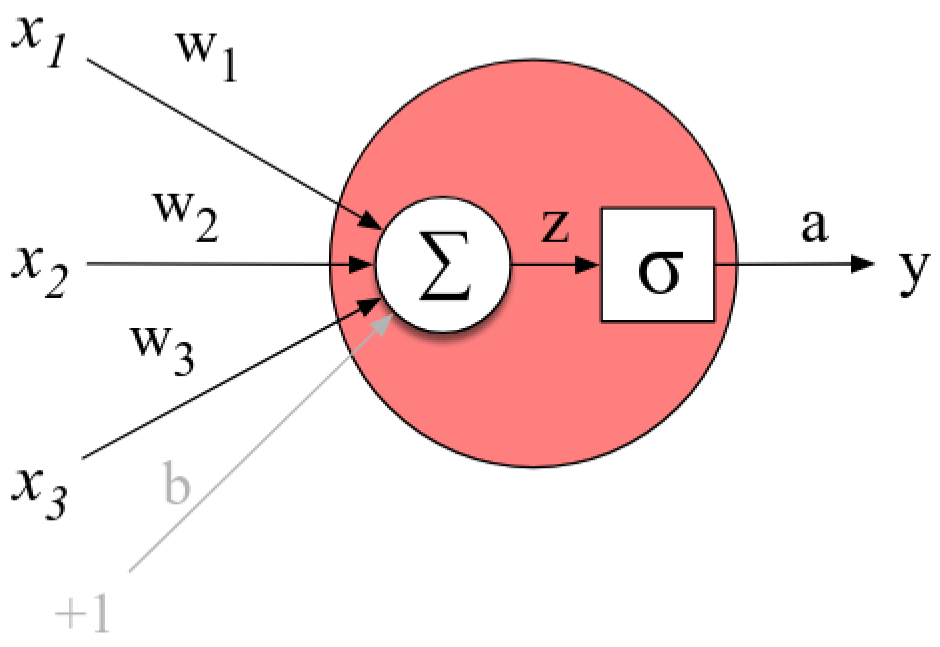
$$z = \mathbf{w} \cdot \mathbf{x} + b$$

Finally, instead of using **z**, a linear function of **x**, as the output, neural units apply a non-linear function **f** to **z**. We will refer to the output of this function as the activation value for the unit, **a**.

Since we are just modeling a single unit, the activation for the node is in fact the final output of the network, which we'll generally call **y**. So the value **y** is defined as:

$$y = a = f(z)$$

$$y = \sigma(z) = \frac{1}{1 + e^{-z}}$$



Activation Functions

We have already covered Sigmoid Function. The other two most commonly used functions are:

Tanh

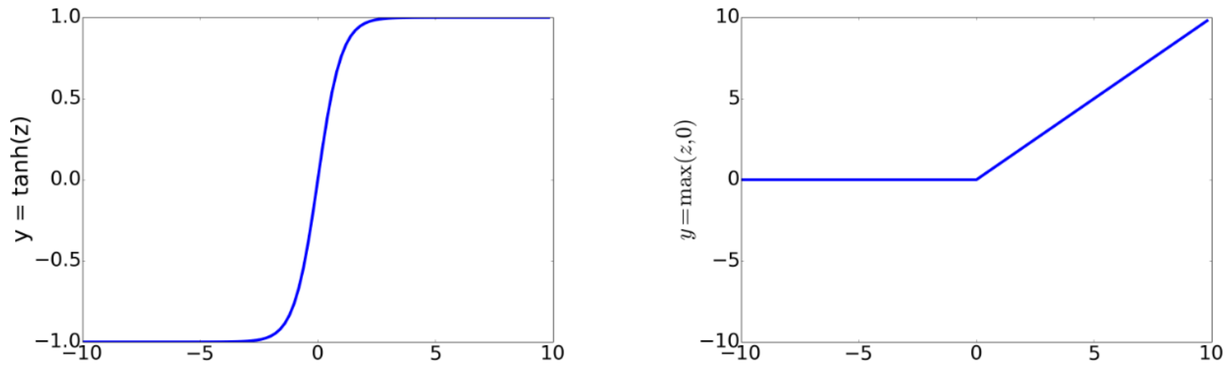
A function that is very similar but almost always better than sigmoid is the tanh function. tanh is a variant of the sigmoid that ranges from -1 to +1. The tanh function has the nice properties of being smoothly differentiable and mapping outlier values toward the mean.

$$y = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

ReLU

The simplest activation function, and perhaps the most commonly used, is the rectified linear unit, also called the ReLU. It’s just the same as **z** when **z** is positive, and 0 otherwise. The rectifier function has nice properties that result from it being very close to linear.

$$y = \text{ReLU(z)} = max(z, 0)$$



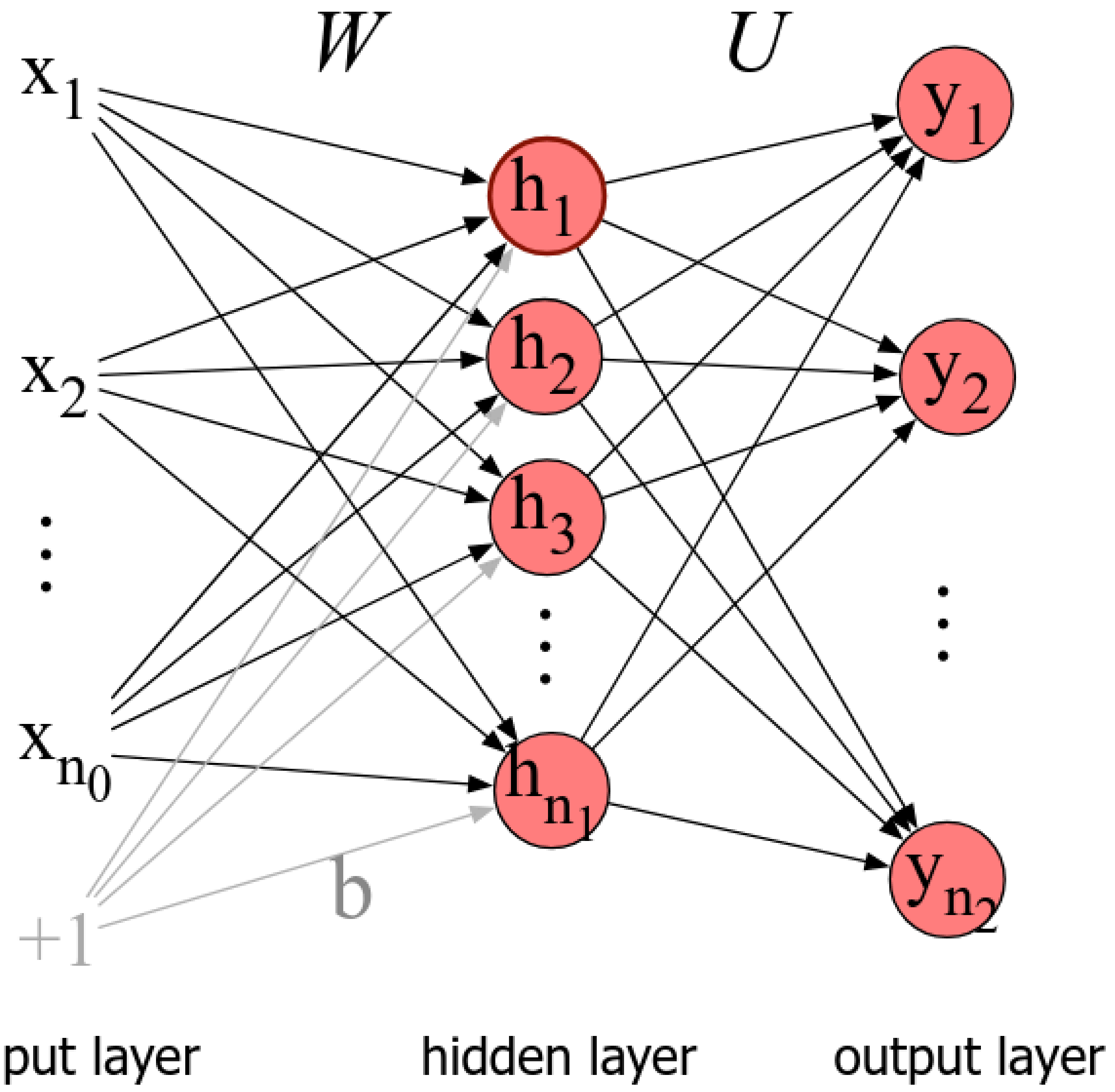
Feedforward Neural Networks

Simple feedforward networks have three kinds of nodes: input units, hidden units, and output units.

The input layer **x** is a vector of simple scalar values.

The core of the neural network is the hidden layer **h** formed of hidden units **h_i**, hidden layer each of which is a neural unit taking a weighted sum of its inputs and then applying a non-linearity. In the standard architecture, each layer is fully-connected, meaning that each unit in each layer takes as input the outputs fully-connected from all the units in the previous layer, and there is a link between every pair of units from two adjacent layers. Thus each hidden unit sums over all the input units.

A single hidden unit has as parameters a weight vector and a bias. We represent the parameters for the entire hidden layer by combining the weight vector and bias for each unit **i** into a single weight matrix **W** and a single bias vector **b** for the whole layer. Each element $\mathbf{W}_{ji}$ of the weight matrix **W** represents the weight of the connection from the ith input unit $\mathbf{x}_i$ to the jth hidden unit $\mathbf{h}_j$.



The computation only has three steps: multiplying the weight matrix by the input vector **x**, adding the bias vector **b**, and applying the activation function **g**.

The output of the hidden layer, the vector **h** is:

$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b})$$

Notice that we're applying the **σ** function here to a vector. We're thus allowing **σ(·)**, and indeed any activation function **g(·)**, to apply to a vector element-wise, so **$g[z_1, z_2, z_3] = [g(z_1), g(z_2), g(z_3)]$**.

We'll refer to the input layer as layer 0 of the network, and have **n_0** represent the number of inputs, so **x** is a vector of real numbers of dimension **n_0**, or more formally **$x \in R^{n0}$** , a column vector of dimensionality **$[n_0, 1]$**. Let's call the hidden layer layer 1 and the output layer layer 2. The hidden layer has dimensionality **n_1**, so **$h \in R^{n1}$** and also **$b \in R^{n1}$** (since each hidden unit can take a different bias value). And the weight matrix **W** has dimensionality **$W \in R^{n1 \times n0}$** , i.e. **$[n_1, n_0]$**.

Like the hidden layer, the output layer has a weight matrix (let's call it **U**), but some models don't include a bias vector **b** in the output layer, so we'll simplify by eliminating the bias vector. The weight matrix is multiplied by its input vector (**h**) to produce the intermediate output **z**:

$$\mathbf{z} = \mathbf{U}\mathbf{h}$$

There are **n_2** output nodes, so **$z \in R^{n2}$** , weight matrix **U** has dimensionality **$U \in R^{n2 \times n1}$** , and element **U_{ij}** is the weight from unit **j** in the hidden layer to unit **i** in the output layer.

However, **z** can't be the output of the classifier, since it's a vector of real-valued numbers, while what we need for classification is a vector of probabilities. There is a convenient function for normalizing a vector of real values, by which we mean converting it to a vector that encodes a probability distribution (all the numbers lie between 0 and 1 and sum to 1): the **softmax** function. More generally for any vector **z** of dimensionality **d**, the softmax is defined as:

$$\text{softmax}(\mathbf{z}_i) = \frac{\exp(\mathbf{z}_i)}{\sum_{j=1}^{d} \exp(\mathbf{z}_j)} \quad 1 \le i \le d$$

Recap

Here are the final equations for a feedforward network with a single hidden layer, which takes an input vector **x**, outputs a probability distribution **y**, and is parameterized by weight matrices **W** and **U** and a bias vector **b**:

$$\begin{aligned}\mathbf{h} &= \sigma(\mathbf{W}\mathbf{x}+\mathbf{b})\\ \mathbf{z} &= \mathbf{U}\mathbf{h}\\ \mathbf{y} &= \mathrm{softmax}(\mathbf{z})\end{aligned}$$

And just to remember the shapes of all our variables, $\mathbf{x} \in \mathbf{R}^{n0}$, $\mathbf{h} \in \mathbf{R}^{n1}$, $\mathbf{b} \in \mathbf{R}^{n1}$, $\mathbf{W} \in \mathbf{R}^{n1\times n0}$, $\mathbf{U} \in \mathbf{R}^{n2\times n1}$, and the output vector $\mathbf{y} \in \mathbf{R}^{n2}$.

Training Neural Nets

Loss Function

The cross-entropy loss that is used in neural networks is the same one we saw for logistic regression. If the neural network is being used as a binary classifier, with the sigmoid at the final layer, the loss function is the same logistic regression loss.

$$L_{CE}(\hat{y},y) = -\log p(y|x) \;=\; -[y\log\hat{y}+(1-y)\log(1-\hat{y})]$$

When we have more than 2 classes we'll need to represent both **y** and **ˆy** as vectors. Let's assume we're doing hard classification, where only one class is the correct one. The true label **y** is then a vector with **K** elements, each corresponding to a class, with $\mathbf{y_c = 1}$ if the correct class is **c**, with all other elements of **y** being 0. Recall that a vector like this, with one value equal to 1 and the rest 0, is called a one-hot vector. And our classifier will produce an estimate vector with **K** elements **ˆy**, each element $\mathbf{\hat{y}_k}$ of which represents the estimated probability $\mathbf{p(y_k = 1|x)}$.

The loss function for a single example **x** is the negative sum of the logs of the **K** output classes, each weighted by their probability $\mathbf{y_k}$:

$$L_{CE}(\hat{\mathbf{y}},\mathbf{y}) = -\sum_{k=1}^{K}\mathbf{y}_k\log\hat{\mathbf{y}}_k$$

the cross-entropy loss is simply the negative log of the output probability corresponding to the correct class, and we therefore also call this the negative log likelihood loss:

$$L_{CE}(\hat{\mathbf{y}},\mathbf{y}) \;=\; -\log\hat{\mathbf{y}}_c \quad \text{(where } c \text{ is the correct class)}$$

Plugging in the softmax formula, and with **K** the number of classes:

$$L_{CE}(\hat{\mathbf{y}},\mathbf{y}) \;=\; -\log \frac{\exp(\mathbf{z}_c)}{\sum_{j=1}^{K}\exp(\mathbf{z}_j)} \quad \text{(where } c \text{ is the correct class)}$$

Computing the Gradient

Computing the gradient requires the partial derivative of the loss function with respect to each parameter. For a network with one weight layer and sigmoid output (which is what logistic regression is), we could simply use the derivative of the loss that we used for logistic regression.

$$\begin{aligned}
\frac{\partial L_{CE}(\hat{\mathbf{y}},\mathbf{y})}{\partial w_j} &= (\hat{y}-y)\,\mathbf{x}_j \\
&= (\sigma(\mathbf{w}\cdot\mathbf{x}+b)-y)\,\mathbf{x}_j
\end{aligned}$$

Or for a network with one weight layer and softmax output (=multinomial logistic regression), we could use the derivative of the softmax loss for a particular weight $\mathbf{w_k}$ and input $\mathbf{x_i}$

$$\begin{aligned}
\frac{\partial L_{\text{CE}}(\hat{\mathbf{y}},\mathbf{y})}{\partial \mathbf{w}_{k,i}} &= -(\mathbf{y}_k-\hat{\mathbf{y}}_k)\mathbf{x}_i \\
&= -(\mathbf{y}_k-p(\mathbf{y}_k=1|\mathbf{x}))\mathbf{x}_i \\
&= -\left(\mathbf{y}_k-\frac{\exp(\mathbf{w_k}\cdot\mathbf{x}+b_k)}{\sum_{j=1}^{K}\exp(\mathbf{w_j}\cdot\mathbf{x}+b_j)}\right)\mathbf{x}_i
\end{aligned}$$

But these derivatives only give correct updates for one weight layer: the last one! For deep networks, computing the gradients for each weight is much more complex, since we are computing the derivative with respect to weight parameters that appear all the way back in the very early layers of the network, even though the loss is computed only at the very end of the network.

The solution to computing this gradient is an algorithm called error backpropagation or backprop.

For the backward pass we'll also need to compute the loss **L**. The loss function for binary sigmoid output is

$$L_{CE}(\hat{y},y) \; = \; -\left[y\log\hat{y} + (1-y)\log(1-\hat{y})\right]$$

Our output **ˆy = a$^{[2]}$**, so we can rephrase this as

$$L_{CE}(a^{[2]},y) \; = \; -\left[y\log a^{[2]} + (1-y)\log(1-a^{[2]})\right]$$

In order to do the backward pass, we'll need to know the derivatives of all the functions. We already saw the derivative of the sigmoid σ :

$$\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$$

We'll give the start of the computation, computing the derivative of the loss function **L** with respect to **z**, or *∂* **L / *∂* z**. By the chain rule:

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial z}$$

So let's first compute *∂* **L / *∂* a$^{[2]}$**

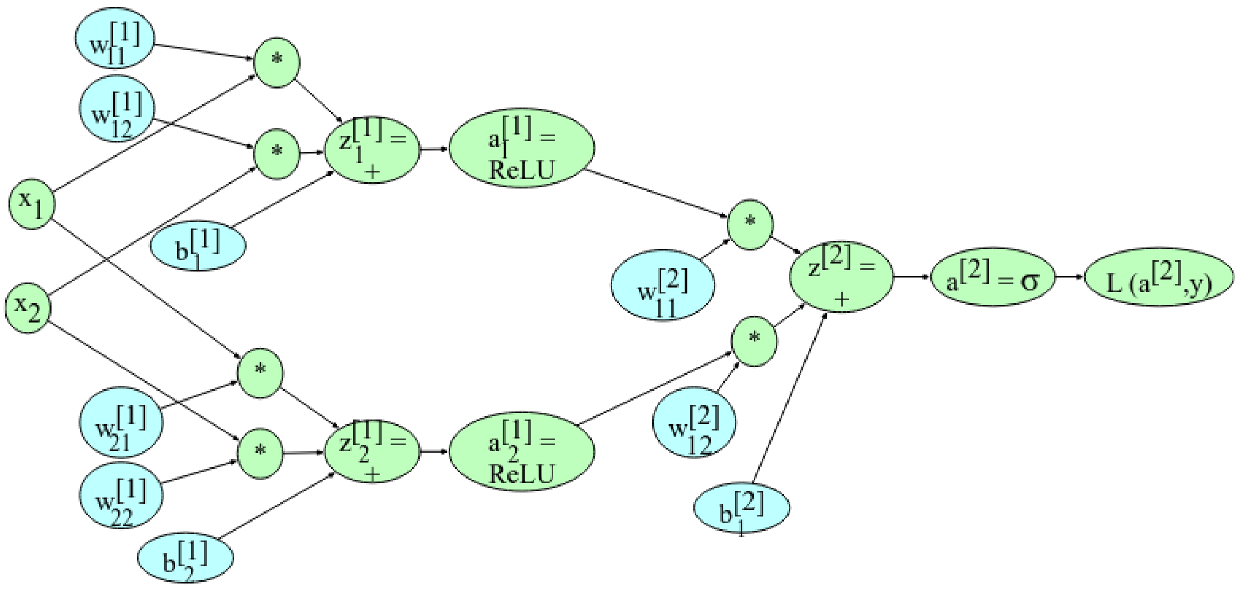
$$\begin{aligned}
L_{CE}(a^{[2]},y) \; &= \; -\left[y\log a^{[2]} + (1-y)\log(1-a^{[2]})\right] \\
\frac{\partial L}{\partial a^{[2]}} \; &= \; -\left(\left(y\frac{\partial\log(a^{[2]})}{\partial a^{[2]}}\right) + (1-y)\frac{\partial\log(1-a^{[2]})}{\partial a^{[2]}}\right) \\
&= \; -\left(\left(y\frac{1}{a^{[2]}}\right) + (1-y)\frac{1}{1-a^{[2]}}(-1)\right) \\
&= \; -\left(\frac{y}{a^{[2]}} + \frac{y-1}{1-a^{[2]}}\right)
\end{aligned}$$

Next, by the derivative of the sigmoid

$$\frac{\partial a^{[2]}}{\partial z} = a^{[2]}(1 - a^{[2]})$$

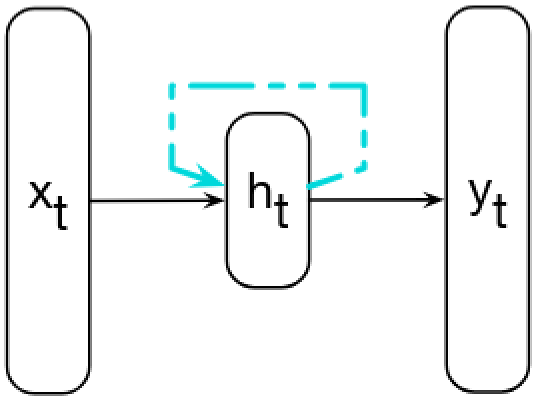
Finally, we can use the chain rule

$$\begin{aligned}\frac{\partial L}{\partial z} &= \frac{\partial L}{\partial a^{[2]}} \frac{\partial a^{[2]}}{\partial z} \\ &= -\left(\frac{y}{a^{[2]}} + \frac{y-1}{1-a^{[2]}}\right) a^{[2]}(1 - a^{[2]}) \\ &= a^{[2]} - y\end{aligned}$$



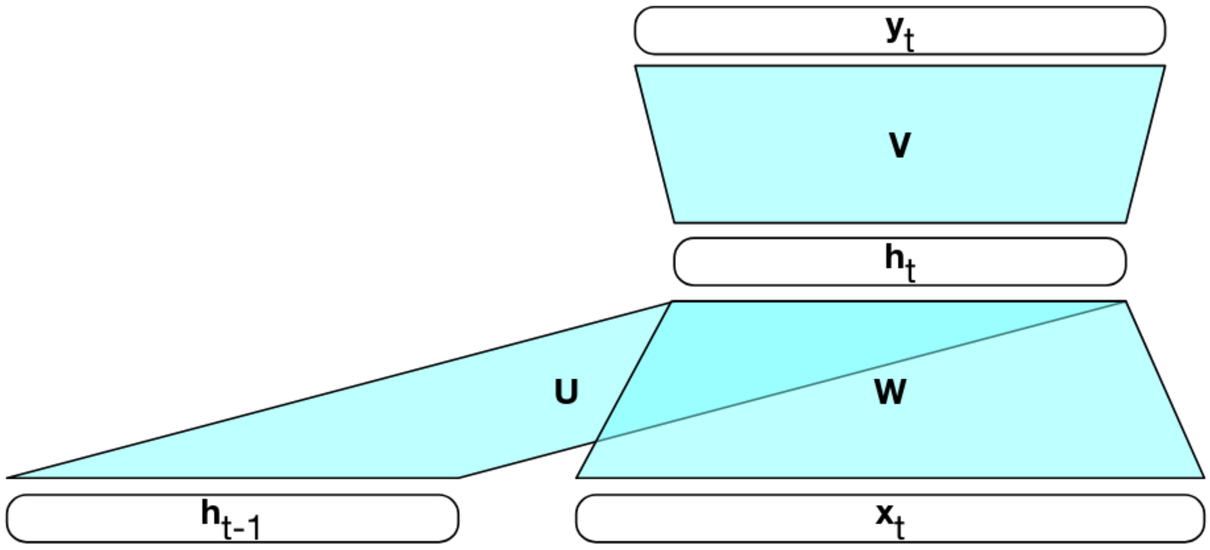
Recurrent Neural Network

A recurrent neural network (RNN) is any network that contains a cycle within its network connections, meaning that the value of some unit is directly, or indirectly, dependent on its own earlier outputs as an input.



Inference

Forward inference (mapping a sequence of inputs to a sequence of outputs) in an RNN is nearly identical to what we've already seen with feedforward networks. To compute an output $\mathbf{y_t}$ for an input $\mathbf{x_t}$, we need the activation value for the hidden layer $\mathbf{h_t}$. To calculate this, we multiply the input $\mathbf{x_t}$ with the weight matrix **W**, and the hidden layer from the previous time step $\mathbf{h_{t-1}}$ with the weight matrix **U**. We add these values together and pass them through a suitable activation function, **g**, to arrive at the activation value for the current hidden layer, $\mathbf{h_t}$. Once we have the values for the hidden layer, we proceed with the usual computation to generate the output vector.



$$
\begin{aligned}
\mathbf{h}_t &= g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{x}_t) \\
\mathbf{y}_t &= f(\mathbf{V}\mathbf{h}_t)
\end{aligned}
$$

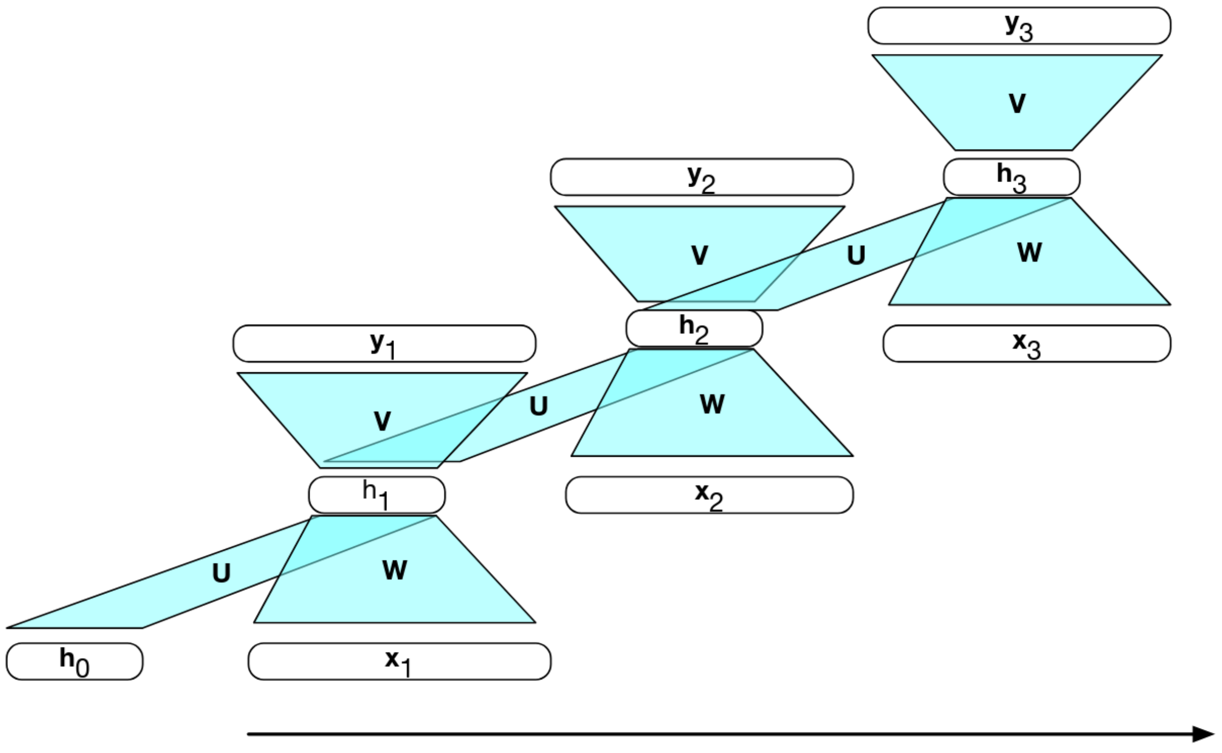
$$\mathbf{y}_t = \text{softmax}(\mathbf{V}\mathbf{h}_t)$$

Let's refer to the input, hidden and output layer dimensions as $\mathbf{d}_{in}$, $\mathbf{d}_h$, and **dout** respectively. Given this, our three parameter matrices are: $\mathbf{W} \in \mathbf{R}^{dh \times din}$, $\mathbf{U} \in \mathbf{R}^{dh \times dh}$, and $\mathbf{V} \in \mathbf{R}^{dout \times dh}$.

The fact that the computation at time **t** requires the value of the hidden layer from time **t − 1** mandates an incremental inference algorithm that proceeds from the start of the sequence to the end.

```
function ForwardRNN($\mathbf{x}$, $network$) returns output sequence $\mathbf{y}$

  $\mathbf{h}_0 \leftarrow 0$
  for $i \leftarrow 1$ to Length($\mathbf{x}$) do
      $\mathbf{h}_i \leftarrow g(\mathbf{U}\mathbf{h}_{i-1} + \mathbf{W}\mathbf{x}_i)$
      $\mathbf{y}_i \leftarrow f(\mathbf{V}\mathbf{h}_i)$
  return $y$
```



$$
\begin{aligned}
\mathbf{e}_t &= \mathbf{E}\mathbf{x}_t \\
\mathbf{h}_t &= g(\mathbf{U}\mathbf{h}_{t-1} + \mathbf{W}\mathbf{e}_t) \\
\mathbf{y}_t &= \mathrm{softmax}(\mathbf{V}\mathbf{h}_t)
\end{aligned}
$$

Training

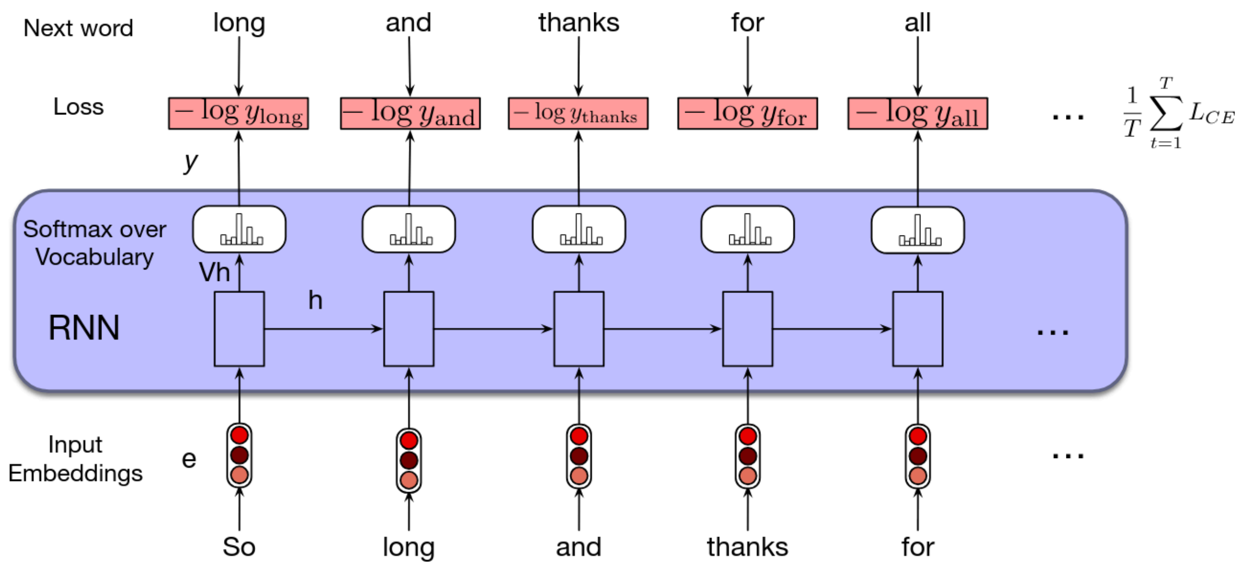
As with feedforward networks, we'll use a training set, a loss function, and backpropagation to obtain the gradients needed to adjust the weights in these recurrent networks. We now have 3 sets of weights to update: **W**, the weights from the input layer to the hidden layer, **U**, the weights from the previous hidden layer to the current hidden layer, and finally **V**, the weights from the hidden layer to the output layer.

To train an RNN as a language model, we use the same self-supervision algorithm we saw earlier: we take a corpus of text as training material and at each time step **t** we have the model predict the next word. We call such a model self-supervised because we don't have to add any special gold labels to the data; the natural sequence of words is its own supervision! We simply train the model to minimize the error in predicting the true next word in the training sequence, using cross-entropy as the loss function. Recall that the cross-entropy loss measures the difference between a predicted probability distribution and the correct distribution.

$$L_{CE} = -\sum_{w \in V} \mathbf{y}_t[w] \log \hat{\mathbf{y}}_t[w]$$

In the case of language modeling, the correct distribution $\mathbf{y}_t$ comes from knowing the next word. This is represented as a one-hot vector corresponding to the vocabulary where the entry for the actual next word is 1, and all the other entries are 0. Thus, the cross-entropy loss for language modeling is determined by the probability the model assigns to the correct next word. So at time **t** the CE loss is the negative log probability the model assigns to the next word in the training sequence.

$$L_{CE}(\hat{\mathbf{y}}_t, \mathbf{y}_t) = -\log \hat{\mathbf{y}}_t[w_{t+1}]$$



LSTMs

The most commonly used such extension to RNNs is the long short-term memory (LSTM) network LSTMs divide the context management problem into two subproblems: removing information no longer needed from the context, and adding information likely to be needed for later decision making. LSTMs accomplish this by first adding an explicit context layer to the architecture (in addition to the usual recurrent hidden layer), and through the use of specialized neural units that make use of gates to control the flow of information into and out of the units that comprise the network layers. These gates are implemented through the use of additional weights that operate sequentially on the input, and previous hidden layer, and previous context layers.

The gates in an LSTM share a common design pattern; each consists of a feed-forward layer, followed by a sigmoid activation function, followed by a pointwise multiplication with the layer being gated.

The first gate we'll consider is the forget gate. The purpose of this gate is to delete information from the context that is no longer needed. The forget gate computes a weighted sum of the previous state's hidden layer and the current input and passes that through a sigmoid. This mask is then multiplied element-wise by the context vector to remove the information from context that is no longer required. Element-wise multiplication of two vectors (represented by the operator , and sometimes called the Hadamard product) is the vector of the same dimension as the two input vectors, where each element i is the product of element i in the two input vectors:

$$\begin{aligned}\mathbf{f}_t &= \sigma(\mathbf{U}_f\mathbf{h}_{t-1} + \mathbf{W}_f\mathbf{x}_t)\\ \mathbf{k}_t &= \mathbf{c}_{t-1} \odot \mathbf{f}_t\end{aligned}$$

The next task is to compute the actual information we need to extract from the previous hidden state and current inputs—the same basic computation we've been using for all our recurrent networks.

$$\mathbf{g}_t = \tanh(\mathbf{U}_g\mathbf{h}_{t-1} + \mathbf{W}_g\mathbf{x}_t)$$

Next, we generate the mask for the add gate to select the information to add to theadd gate current context.

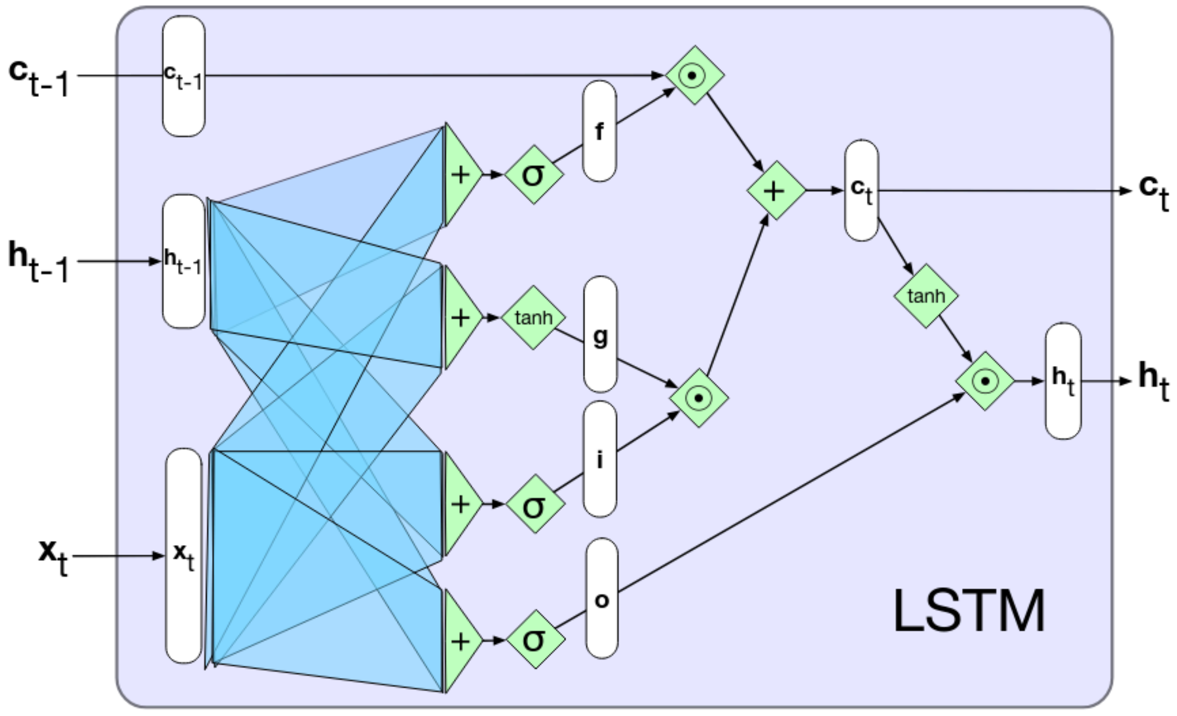
$$\begin{aligned}\mathbf{i}_t &= \sigma(\mathbf{U}_i\mathbf{h}_{t-1} + \mathbf{W}_i\mathbf{x}_t)\\ \mathbf{j}_t &= \mathbf{g}_t \odot \mathbf{i}_t\end{aligned}$$

Next, we add this to the modified context vector to get our new context vector.

$$\mathbf{c}_t = \mathbf{j}_t + \mathbf{k}_t$$

The final gate we'll use is the output gate which is used to decide what informa-output gate tion is required for the current hidden state (as opposed to what information needs to be preserved for future decisions).

$$\begin{aligned}\mathbf{o}_t &= \sigma(\mathbf{U}_o\mathbf{h}_{t-1} + \mathbf{W}_o\mathbf{x}_t) \\ \mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)\end{aligned}$$



Task 1

Create a Feedforward Neural Network from scratch without any 3rd party libraries for sentiment analysis.

Task 2

Create a Recurrent Neural Network from scratch without any 3rd party libraries for sentiment analysis.

Task 3

Create an LSTM from scratch without any 3rd party libraries for sentiment analysis.